C# 프로그래밍

- 5 주차 (1강)

SEJONG UNIVERSITY

객체지향 프로그래밍

- 은닉성 (캡슐화)
- 상속성
- 다형성

SEJONG UNIVERSITY

Introduction to 상속 (Inheritance)

- 현실 세계에서 많은 객체가 계층적 관계를 따름
  - ✓ 한 객체가 다수의 다른 객체의 공통된 특성을 가지고 있음
- 예를 들어, 노트북, 데스크톱, 넷북이라는 타입을 정의
  - ✓ 공통적으로 부팅, 전원 끄기, 리셋 등 컴퓨터로의 공통된 동작 수행
  - ✓ 상속이라는 개념이 없으면, 각각 개별적으로 메서드와 상태 값 정의

```
public class Notebook
{
    bool powerOn;
    public void Boot() { }
    public void Shutdown() { }
    public void Reset() { }

    bool fingerScan;
    public bool HasFingerScanDevice()
    {
        return fingerScan;
    }
}
```

```
public class Desktop
{
    bool powerOn;
    public void Boot() { }
    public void Shutdown() { }
    public void Reset() { }
}
```

```
public class Netbook
{
    bool powerOn;
    public void Boot() { }
    public void Shutdown() { }
    public void Reset() { }
}
```

상속 (Inheritance)

- 클래스는 다른 클래스로부터 유산을 물려받을 수 있음
  - ✓ 즉, 필드나 메소드, 프로퍼티 같은 멤버를 물려받음
  - ✓ 상속하는 쪽을 부모(또는 기반) 클래스, 상속 받는 쪽을 자식(또는 파생) 클래스
  - ✓ 클래스 이름 옆에 콜론(:) 과 상속받을 클래스 이름 작성
  - ✓ "protected" 접근 제한자는 클래스 멤버를 외부에 접근 차단하면서 자식에게는 허용

```
public class Computer
{
    protected bool powerOn;
    public void Boot() { }
    public void Shutdown() { }
    public void Reset() { }
}

public class Desktop : Computer { }

public class Netbook : Computer { }
```

```
public class Notebook : Computer
{
    bool fingerScan;
    public bool HasFingerScanDevice(){ return fingerScan; }

    public void CloseLid()
    {
        if (powerOn == true)
        {
            Shutdown();
        }
    }
}
```

상속이 불가능 한 경우

- C# 은 단일 상속만 지원
  - ✓ 동시에 둘 이상의 부모 클래스로부터 다중 상속은 허용하지 않음
- sealed 키워드를 이용해 상속이 불가능 하도록 클래스를 선언
  - ✓ 의도하지 않은 상속 또는 파생 클래스 구현을 막음
  - ✓ 메소드 오버라이딩 제한

다중 상속 지원불가

```
public class Computer { }
public class Monitor { }

// 컴파일 에러 발생
public class Notebook : Computer, Monitor
{

}
```

의도치 않은 상속

```
sealed class Monitor { }

// 컴파일 에러 발생
public class Notebook : Monitor
{

}
```

오버라이딩 제한

```
class Parent
{
    public virtual void Test() { }
}

class Child : Parent
{
    public sealed override void Test() { }
}

class GrandChild : Child
{ // 컴파일 에러 발생
    public override void Test() { }
}
```

부모/자식 클래스 사이의 형식 변환

- 정수형 변수 사이의 형식 변환
  - ✓ 범위가 작은 데이터 타입에서 큰 타입으로는 암시적 변환
  - ✓ 범위가 큰 데이터 타입에서 작은 타입으로는 명시적 변환
- 부모/자식 클래스 사이의 형식 변환
  - ✓ 자식 클래스의 객체를 부모 클래스의 객체로 대입은 암시적 변환
  - ✓ 부모 클래스에서 자식 클래스로의 변환은 명시적 변환
    - 컴파일 가능하지만 경우에 따라 **런타임 에러 발생 할 수 있음**

```
sbyte a = 127;
int b = a;            // 암시적 형변환
int c = 50;
sbyte d = (sbyte)c;   // 명시적 형변환
```

정수형 사이의 형변환

클래스 사이의 암시적/명시적 변환

```
Notebook note1 = new Notebook();
// 암시적 형변환 가능
Computer pc1 = note1;
pc1.Boot();
pc1.Shutdown();

Computer pc2 = new Computer();
// 명시적 형변환, 컴파일은 가능, but 런타임 에러 가능
Notebook note2 = (Notebook)pc2;
```

C# 에서 클래스의 명시적 변환 지원 이유

```
Notebook note3 = new Notebook();
// 부모 타입으로 암시적 형변환
Computer pc3 = note3;

// 다시 본래 타입으로 명시적 형변환
Notebook note4 = (Notebook)pc3;
note4.CloseLid();
```

부모/자식 클래스 형변환 활용 (1)

- 암시적 형변환이 명시적 형변환보다 자주 사용
  - ✓ 메소드 오버로딩 사용 때 보다 코드의 생산성 향상

형변환을 이용해서 메소드를 작성한 경우

```
public class DeviceManager
{
    public void TurnOff(Computer device)
    {
        device.Shutdown();
    }
}
```

오버로딩을 이용해서 메소드를 작성한 경우

```
public class DeviceManager
{
    public void TurnOff(Notebook notebook) {/* ... */}
    public void TurnOff(Desktop desktop) {/* ... */}
    public void TurnOff(Netbook netbook) {/* ... */}
}
```

동작 확인을 위한 메소드 호출

```
static void Main(string[] args)
{
    Notebook notebook = new Notebook();
    Desktop desktop = new Desktop();
    Netbook netbook = new Netbook();

    DeviceManager manager = new DeviceManager();
    manager.TurnOff(notebook);
    manager.TurnOff(desktop);
    manager.TurnOff(netbook);
}
```

부모/자식 클래스 형변환 활용 (2)

- 암시적 형변환을 통해 자식 클래스의 객체를 부모 객체의 배열에 저장

형변환을 이용해서 메소드를 작성한 경우

```
public class DeviceManager
{
    public void TurnOff(Computer device)
    {
        device.Shutdown();
    }
}
```

부모 객체 배열 이용

```
Computer[] machines = new Computer[] { new Notebook(), new Desktop(), new Netbook() };
// 암시적 형변환

DeviceManager manager = new DeviceManager();
foreach (Computer device in machines)
{
    manager.TurnOff(device);
}
```

as, is 연산자

연산자	설명
as	형식 변환 연산자와 같은 역할. 다만 **형식 변환 연산자가 변환에 실패하는 경우 예외를 던지는 반면 as 연산자는 객체 참조를 null 로 만든**다는 차이가 있음.
is	객체가 해당 형식에 해당하는지 검사하여 결과를 bool 값으로 반환.

```
class Mammal
{
    public void Nurse() {  }
}

class Dog : Mammal
{
    public void Bark() { }
}

class Cat : Mammal
{
    public void Meow() { }
}
```

as 연산자 사용

```
Mammal mammal = new Cat();
Cat cat = mammal as Cat;
// 만약 형식변환 실패했다면 cat은 null
if (cat != null)
{
    cat.Meow();
}
```

is 연산자 사용

```
Mammal mammal2 = new Dog();
Dog dog;

if (mammal2 is Dog)
{ // 객체가 Dog 임을 확인, 안전한 형변환
    dog = (Dog)mammal2;
    dog.Bark();
}
```

as, is 연산자

- 명시적 형식변환 대신 as 연산자 사용을 권장
  - ✓ 형식변환에 실패해도 예외 발생하지 않기 때문에 코드 관리가 수월
- as 연산자는 참조형 변수에 대해서만 적용
- is 연산자는 참조형식 뿐 아니라 값 형식에도 사용 가능

```
int n = 5;
if ((n as string) != null)  // 컴파일 오류 발생
{
    Console.WriteLine("변수 n은 string 타입");
}

string txt = "text";
if ((txt as int) != null)   // 컴파일 오류 발생
{
    Console.WriteLine("변수 txt는 int 타입");
}
```

```
int n = 5;
if (n is string)  // 정상 동작
{
    Console.WriteLine("변수 n은 string 타입");
}

string txt = "text";
if (txt is int)   // 정상 동작
{
    Console.WriteLine("변수 txt는 int 타입");
}
```

as, is 연산자 예제 코드

```
class Mammal
{
    public void Nurse()
    {
        Console.WriteLine("Nurse()");
    }
}

class Dog : Mammal
{
    public void Bark()
    {
        Console.WriteLine("Bark()");
    }
}

class Cat : Mammal
{
    public void Meow()
    {
        Console.WriteLine("Meow()");
    }
}
```

```
static void Main(string[] args)
{

    Mammal mammal = new Dog();
    Dog dog;

    if (mammal is Dog)
    { // 객체가 Dog 임을 확인, 안전한 형변환
        dog = (Dog)mammal;
        dog.Bark();
    }

    Mammal mammal2 = new Cat();
    Cat cat = mammal2 as Cat;
    // 만약 형식변환 실패했다면 cat은 null
    if (cat != null)
    {
        cat.Meow();
    }

    Cat cat2 = mammal as Cat;
    if (cat2 != null)
        cat2.Meow();
    else
        Console.WriteLine("cat2 is not a Cat");
}
```

출력 결과:

```
Bark()
Meow()
cat2 is not a Cat
```

부모/자식 클래스 생성자와 종료자 호출

- 자식 클래스는 객체를 생성할 때 부모 클래스의 생성자를 호출 후에 자신의 생성자 호출
- 객체가 소멸할 때 자신의 종료자 호출 후 부모 클래스 종료자 호출

```
class Base
{
    public Base() {
        Console.WriteLine("Base()"); }

    ~Base() {
        Console.WriteLine("~Base()"); }
}

class Derived : Base
{
    public Derived() {
        Console.WriteLine("Derived()"); }

    ~Derived() {
        Console.WriteLine("~Derived()"); }
}
```

```
class MainApp
{
    public static void Test()
    {
        Derived derived = new Derived();
    }

    static void Main(string[] args)
    {
        Test();
        // 가비지 컬렉터 수행
        GC.Collect ();
        // 종료자 큐 처리동안 wait
        GC.WaitForPendingFinalizers();
    }
}
```

출력 결과:
Base()
Derived()
~Derived()
~Base()

상속받는 경우 생성자로 인한 오류

- 자식 클래스는 객체를 생성할 때 부모 클래스의 생성자를 자동 호출
- 자동 호출되는 부모 클래스의 생성자가 매개변수를 가지고 있지만 입력 값을 전달하지 못할 때 에러 발생
- base 키워드
  - ✓ 부모 클래스를 가리킴

base 키워드 사용

```
class Base
{
    public void BaseMethod()
    {/* ... */}
}

class Derived : Base
{
    public Derived()
    {
        base.BaseMethod();
    }
}
```

생성자 오류

```
class Book
{
    decimal isbn13;
    public Book(decimal isbn13)
    {
        this.isbn13 = isbn13;
    }
}

class EBook : Book
{
    public EBook()  // 컴파일 에러 발생
    {

    }
}
```

base() 생성자

- base() 는 부모 클래스의 생성자를 나타냄
- base() 에 매개변수를 넘겨 호출
  - ✓ 매개변수를 기본 값으로 초기화
  - ✓ 자식 클래스의 생성자에 입력 받은 매개변수 값을 base() 로 연계하여 사용

```
class Book
{
    decimal isbn13;
    public Book(decimal isbn13)
    {
        this.isbn13 = isbn13;
    }
}

class EBook : Book
{
    public EBook() : base(0)
    {
    }

    public EBook(decimal isbn) : base(isbn)
    { // 매개변수 값을 연계하여 사용
    }
}
```

base() 생성자 예제코드

```
class Base
{
    protected string Name;
    public Base(string Name)
    {
        this.Name = Name;
        Console.WriteLine($"{this.Name}.Base()");
    }

    ~Base()
    {
        Console.WriteLine($"{this.Name}.~Base()");
    }

    public void BaseMethod()
    {
        Console.WriteLine($"{this.Name}.BaseMethod()");
    }
}
```

```
class Derived : Base
{
    public Derived(string Name) : base(Name) {
        Console.WriteLine($"{this.Name}.Derived()");
    }

    ~Derived() {
        Console.WriteLine($"{this.Name}.~Derived()");
    }

    public void DerivedMethod() {
        Console.WriteLine($"{this.Name}.DerivedMethod()");
    }
}
```

메인 메소드 코드

```
public static void Test()
{
    Base a = new Base("a");
    a.BaseMethod();
    Derived b = new Derived("b");
    b.BaseMethod();
    b.DerivedMethod();
}
static void Main(string[] args)
{
    Test();
    GC.Collect();
    GC.WaitForPendingFinalizers();
}
```

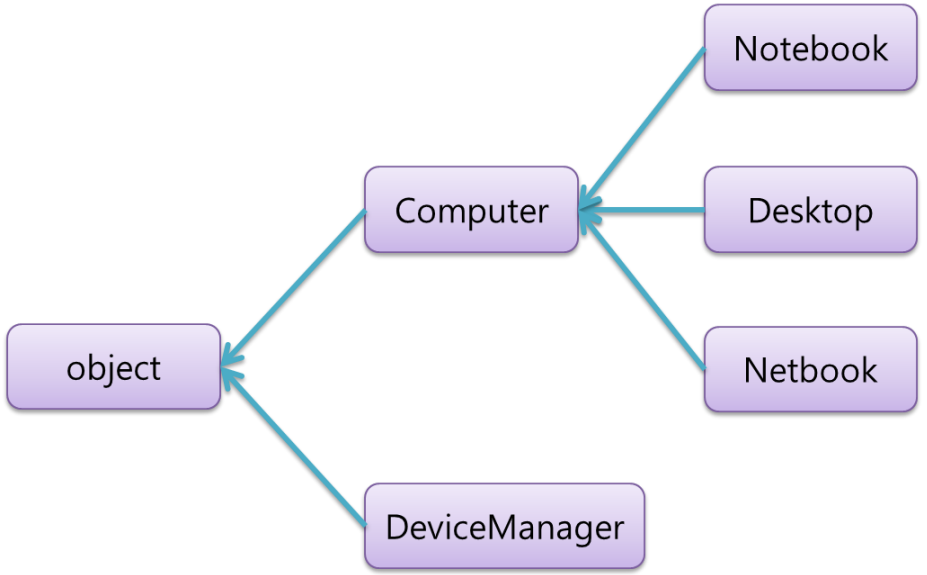
출력 결과:

```
a.Base()
a.BaseMethod()
b.Base()
b.Derived()
b.BaseMethod()
b.DerivedMethod()
b.~Derived()
b.~Base()
a.~Base()
```

모든 클래스의 부모: object

- C# 에서 정의되는 모든 클래스의 부모는 object
- object 클래스는 아래 네 개의 public 메소드 포함

```
public class Object
{
    public virtual bool Equals();
    public virtual int GetHashCode();
    public Type GetType();
    public virtual string ToString();
}
```



object 객체와의 형식 변환

```
Computer computer = new Computer();
object obj1 = computer;
Computer pc1 = obj1 as Computer;

Notebook notebook = new Notebook();
object obj2 = notebook;
Notebook pc2 = obj2 as Notebook;
```

Thank you!